# MANAGING XML DATA: AN ABRIDGED OVERVIEW

*XML's flexibility makes it a natural format for both exchanging and integrating data from diverse data sources. In this survey, the authors give an overview of issues in managing XML data, discuss existing solutions, and outline the current technology's open problems and limitations.*

XML's popularity has made it the prime standard for exchanging data on the Web. A diverse set of factors has fueled the explosion of interest in XML (www.w3.org/TR/REC-xml): XML's self-describing nature makes it more amenable for use in loosely coupled data-exchange systems, and the flexible semistructured data model behind it makes it natural as a format for integrating data from various sources. But much of its success stems from the existence of standard languages for each aspect of XML processing and the rapid emergence of tools for manipulating XML. Important related standards include schema languages such as XML Schema (www.w3.org/XML/Schema), which provide notation for defining elements and documents; query languages such as XML Path (XPath; www.w3.org/TR/xpath) and XQuery (www.w3.org/TR/xquery), which provide a means for selecting elements and querying XML documents; and Extensible Stylesheet Language Transformations (XSLT, www.w3.org/TR/xslt), a language for defining transformations of an XML document into different representations and formats. Popular tools include parsers such as Xerces (http://xml.apache.org/xerces-j), query processors such as Galax (http://db.bell-labs.com/galax), and transformation tools such as Xalan (http://xml.apache.org/xalan-j).

The development of this standards framework has made XML dialects powerful vehicles for standardization in communities that exchange data.

In this article, we discuss the main problems involved in managing XML data. Our objective is to clarify potential issues that must be considered when building XML-based applications—in particular, XML solutions' benefits as well as possible pitfalls. Our intent is not to give an exhaustive review of XML data-management (XDM) literature, XML standards, or a detailed study of commercial products. Instead, we aim to provide an overview of a representative subset to illustrate how some XDM problems are addressed.

1521-9615/04/$20.00 © 2004 IEEE
Copublished by the IEEE CS and the AIP

JULIANA FREIRE
*Oregon Health & Science University*
MICHAEL BENEDIKT
*Bell Laboratories, Lucent Technologies*

## Need for XML Data Management Tools and Techniques

As XML data becomes central to applications, there is a growing need for efficient and reliable XDM tools and techniques. Figure 1 illustrates XML's various roles in applications:

- *publishing*—converting non-XML data into XML;
- *storage*—mapping XML data into formats that can be stored in a database; and
- *access*—retrieving, querying, or transforming XML documents, either from storage or streaming in from a network.

To enable data to be exchanged regardless of the platform on which it's stored or the data model in which it is represented, numerous groups publish document type definitions (DTDs) and XML schemata that specify the format of the XML data to be exchanged between their applications. The Cover Pages Web site (http://xml.coverpages.org) contains a comprehensive collection of XML dialects used in a variety of application domains. Because data typically is stored in non-XML database systems, applications must publish data in XML for exchange purposes. When a target application receives XML data, it can remap and store it in internal data structures or a target database system. Applications can also access an XML document either through APIs such as the Document Object Model (DOM; www.w3.org/DOM) or query languages. The applications can directly access the document in native format or, with conversion, from a network stream or non-XML database format.

In contrast with relational database management systems (RDBMSs) that had a clear initial motivation in supporting online transaction processing (OLTP) scenarios, XML applications' requirements vary widely. Applications must deal with several different kinds of queries (structured and keyword-based) in different scenarios (with or without transaction support, over stored or streaming data), as well as data with varying characteristics (ordered and unordered, with or without a schema). The ability to handle widely different scenarios adds significant complexity to various data management tasks. Not surprisingly, XDM is an active area of research. Commercial database vendors have also shown significant interest in XDM—support for XML data is present in most RDBMSs. Examples include IBM's DB2 XML Extender (www4.ibm.com/software/data/db2/extenders/xmlext.html), Microsoft's support for XML (http://msdn.microsoft.com/sqlxml/), and Oracle's XML DB (http://otn.oracle.com/tech/xml/xmldb/).

XML technology, however, is still immature and many of its promises are unfulfilled. This is particularly true for XDM, in which the basic problems of storage, publishing, and querying still lack scalable solutions. In addition, because XML is so flexible and extensible, there is no one-size-fits-all solution, and applications are often required to assemble an infrastructure consisting of several tools.

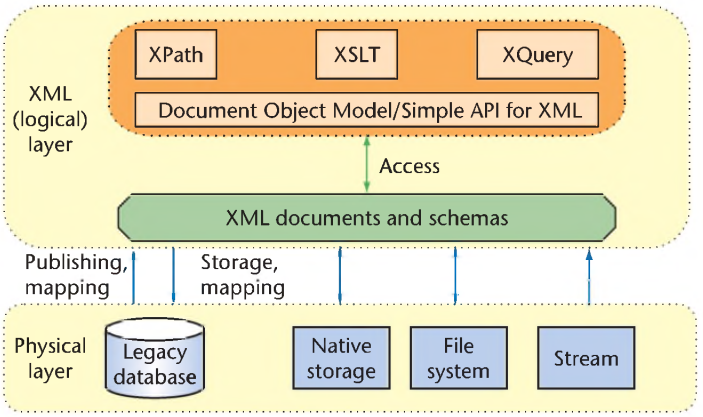


**Figure 1. Problem areas in XML data management. This figure illustrates the different roles XML plays in applications and the corresponding data-management problems: data access, publishing, and storage.**

Consider the following scenario that takes place in the lab of a Dr. Einstein and illustrates XML's many uses.

In his lab, Einstein uses XML dialects to represent data acquired from lab sensors and simulations, metadata about experiments that manipulate these data, and technical reports. Einstein also uses XML to transmit research results to partner labs. Owing to the high throughput of data that streams from the sensors and the numerical nature of simulation data, Einstein uses a specialized storage engine for these data. The metadata describing the experiments, which the lab staff frequently queries and updates, resides in an RDBMS whereas the technical reports are stored in a document management system that provides a keyword-based search interface. Research results documents are built from a mix of raw data and metadata using a publishing tool.

## Data Access Interfaces

An important dimension of the XDM problem is data access. Standards have been defined for a set of APIs and query languages; Figure 2 illustrates how these fit together by drawing an analogy with RDBMSs. At the lowest level of the relational data-access stack, data is stored in disk pages managed by the RDBMS. The relational data model gives an abstract view of the physical storage—a data model—comprised of named tables that contain fields with atomic types. One can access the data using this abstract data model via the programmatic interfaces Java database connectivity (JDBC) and open database connectivity (ODBC); or using declarative query lan-

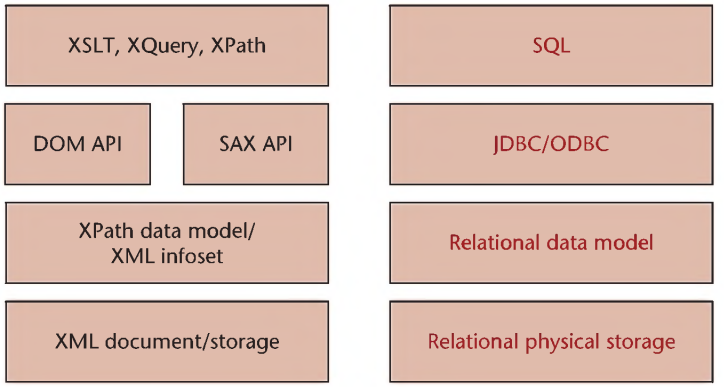


**Figure 2. XML APIs and their relational analogues. The figure illustrates the components of data access in XML-based applications by contrasting them against their relational analogues.**

guages such as SQL. In contrast, the serialized representation of an XML document is at the bottom of the XML application stack. The Information Set (Infoset; www.w3.org/TR/xml-infoset/) and the XPath Data Model (www.w3.org/TR/xpath-datamodel/) define abstract data models for an XML document. The APIs, DOM and Simple API for XML (SAX, www.saxproject.org), provide programmatic interfaces to access the data exported by the Infoset. The query languages XSLT, XQuery, and XPath use the XPath data model as the target model to query the XML document.

### Programmatic APIs

DOM and SAX are language-independent programmatic APIs for accessing the contents of an XML document. XML parsers, such as the previously mentioned Xerces, often support these APIs. DOM provides navigational access to an XML document. After a document is parsed, a DOM instance of the document can be generated, which allows read and write access to the nodes in the document tree and their content; for example, given a node, DOM allows access to its contents as well as access to its children, siblings, and parent. Whereas DOM creates an in-memory representation of an XML document, SAX provides stream-based access to documents. As a document is parsed, events are fired for each open and close tag encountered. Thus, in contrast to DOM, SAX only supports read-once processing of documents.

### Query Languages

In contrast to programmatic interfaces that specify how to access a document's contents, query languages provide a declarative means to do so—that is, they specify what is required. In XML, common querying tasks include filtering and selecting values, merging and integrating values from multiple documents, and transforming XML documents. XPath, XSLT, and XQuery all support these tasks. In the earlier scenario, they could be used to transform the experimental data from Dr. Einstein's internal XML format into the XML exchange format his partner labs require, or to query the experiments' metadata.

XPath is a common language for filtering and selecting values and is used in XSLT, XQuery, and several other languages. XSLT is a loosely typed scripting language whose primary purpose is to transform XML documents into other representations (for example, into HTML for display in a Web browser). Like other browser-oriented technologies, XSLT is designed to be highly tolerant of variability and errors in input data. XQuery, in contrast, is a strongly-typed query language whose primary purpose is to support queries against XML databases. Similar to database query languages, XQuery must guarantee data operations' safety and correctness. Several tools are available to process XML queries; for example, Xalan fully implements XSLT and XPath, and Galax is an open-source reference implementation of XQuery 1.0.

## Storage

Analogous to the relational model, the XML data model provides physical independence. As Figure 1 illustrates, because the different APIs and query languages that access XML documents' contents are based on the XML data model, they are not tied to how the XML data is physically stored. As a result, an XML-based data-management system can be "storage agnostic": it can use and combine many different kinds of storage systems, from custom data structures and RDBMSs to ASCII files.

Because different storage models are possible for XML, an important question is how to select the best alternative for a given application. Critical issues that must be considered when selecting a storage solution include the kind of data to be stored (structured or semistructured); the type of access required for the stored data (which query classes will be posed); and, of course, application requirements (such as support for transactions). We describe benefits and drawbacks of three broad classes of storage alternatives for XML: flat files, colonial solutions, and native systems.

### Files in a File System

Storing XML in the file system (as ASCII files) is an efficient solution if whole documents are stored and

retrieved as a unit. However, for navigational queries, the stream representation has inherent problems in performance and scalability because parsing can become prohibitively expensive for large documents.

### Colonial Solutions

Colonial solutions reuse existing storage systems by mapping XML into the storage system's model. Because the relational model is the most mature in terms of both standards and implementations, the most common approach is to layer XML storage on top of relational engines. Besides the ability to reuse stable, reliable, and efficient systems, the colonial approach allows for simpler integration of the XML data with data that is already stored in these systems; in a large organization that uses RDBMSs, it is possible to manage new XML data in the same environment as the existing data. Mapping XML to existing storage systems, however, is challenging. Figure 3 illustrates the main tasks a user must perform to store XML in relational databases. First, the XML document's schema must be mapped by either a database administrator or a program into an equivalent relational schema. The tree-structured XML documents are then shredded into flat pieces and loaded into the relational tables. Finally, at runtime, XML queries are translated into SQL, submitted to the RDMBS, and the results are then translated into XML.

Due to the mismatch between the XML and the relational models, there are many different ways to map an XML document into relations (relational tables). Most commercial RDBMSs let users manually specify mappings. This approach gives flexibility, but has important drawbacks: users must have knowledge of both XML and relational technologies, and manually defining these mappings is often a lengthy and complex process. Additionally, it is difficult to manually select a mapping that will lead to the best performance from among many different choices, especially because several factors (the query workload, document content) contribute to a given mapping choice's performance. In recent literature, researchers proposed strategies that automate the mapping generation process,[1,2] but these proposals fail to account for application characteristics. Due to XML's flexible infrastructure, different XML applications exhibit widely different characteristics, and a specific fixed mapping strategy is unlikely to perform well for all different applications. The LegoDB system uses a cost-based strategy for mapping XML documents into relations, which automatically generates the most efficient mapping for a given application.[3]

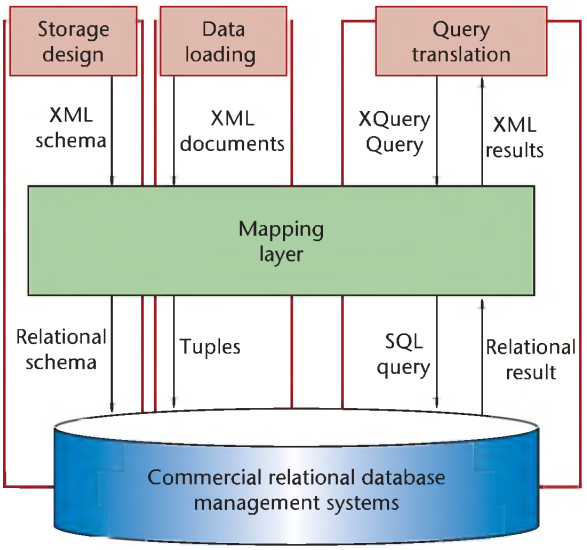


**Figure 3. Storing XML in relational databases. There are three main tasks required to store and query XML in a relational database: storage design, data loading, and query translation.**

### Native

In contrast with the colonial approach, which reuses existing storage systems, native systems are designed with the XML data model and query languages as direct targets. They provide specialized indices,[4,5] specialized query-processing algorithms,[6] data-layout strategies, and recovery and concurrency control.[7] Consequently, native systems often provide better support for XML-specific features. However, implementing a complete system from scratch that supports general data-management features (such as access control, transactions, recovery, and replication) is a large-scale and costly effort. Not surprisingly, native systems often have incomplete support for these features. Several native systems are available, including open source systems such as Xindice (http://xml.apache.org/xindice/), commercial ones such as Tamino (www.softwareag.com/tamino/), and research systems such as Timber[6] and NatiX.[7]

## Publishing

XML lets data be exchanged in a standard format that is independent from how the data is stored: the data being exchanged is an XML view of data that resides and is updated in a non-XML storage system—typically, an RDBMS. In some cases, when this physical representation can be shaped according to the application's performance needs, the full spectrum of storage techniques and mappings is available. However, a common special case is a *publishing* scenario: when some preexisting, independently maintained non-

XML data must be converted to XML. In this case, the data management infrastructure has no control over the physical storage, only the mapping to XML.

For some applications, simply publishing relational data—either entire tables or query results—in some generic XML format suffices. Applications can use the resulting XML document as an intermediate representation that can be queried or transformed, or they can easily integrate it with other data sets with differing but overlapping schemata. In this scenario, publishing is simple because the resulting structure mirrors the original relational tables' flat structure. Commercial relational systems support this facility—for example, Oracle provides the **FOR XML** construct to indicate that an SQL results query should be output in a canonical, flat XML format. The statement "`SELECT projectname, scientist, date, notes FROM LabNotes` **`FOR XML Auto`**" outputs

```
<rowset>
  <row> <projectname> Relativity
        </projectname>
    <scientist> Albert Einstein
        </scientist>
    <date> March, 1918 </date>
    <notes> The special theory of
    relativity has crystallised out
    from the Maxwell-Lorentz theory
    of electromagnetic phenomena.
    Thus all facts of experience
    which support the electromagnetic
    theory also support the theory of
    relativity. </notes> </row>

  <row> <projectname> Relativity
        </projectname>
    <scientist> Albert Einstein
        </scientist>
    <date> December, 1920 </date>
    <notes> Thought experiment: THE
    SURFACE of a marble table is
    spread out in front of me. I can
    get from any one point on this
    table to any other point by pass-
    ing continuously... </notes>
    </row>
...
</rowset>
```

In most cases, however, it is not up to the application to define the XML format in which data must be exported. While XML has enabled the creation of standard data formats within industries and communities, adoption of these standards has led to an enormous and immediate problem of exporting data available in legacy formats to meet newly created standard schemata. For example, scientific data must be wrapped so that it can be exported in the Extensible Data Format (XDF; http://xml.gsfc.nasa.gov/XDF/XDF_home.html). Not only are these standard formats fixed externally, but unlike the canonical (flat) format, they generally feature deeply nested structures that radically differ from the legacy relational representation.

### Publishing Languages

Several publishing languages have been proposed to specify XML views over the legacy data—that is, how to map legacy data (such as tables) into a predefined XML format. IBM's DB2 Extender and SQL Server allow users to annotate an XML Schema with instructions on how to populate the various elements using data in the relational tables. As illustrated below, SQL Server lets users associate tables and fields to XML elements by adding annotations (shown in boldface) to the respective elements. For example, the annotation `sql:relation="LabNotes"` indicates that the element "EinsteinNotes" will be populated with the contents of the table LabNotes.

```
<xs:element name="EinsteinNotes"
       sql:relation="LabNotes" >
  <xs:complexType>
    <xs:sequence>
      <xs:element name="projectname"
       type="xs:string"
       sql:field="projname" />
      <xs:element name="date"
       type="xs:string"
       sql:field="projdate" />
        ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

These annotation-based approaches, however, have limited expressive power. Although they allow elements in the XML template to be associated using selection and join conditions on the relational tables, they do not support arbitrary SQL queries. Hence, it might not be possible to generate complex XML views.

Another approach is to export the relational data in a canonical XML format and use XSLT or XQuery to transform the document into the desired format. Besides the added expressiveness, this approach also leverages the power of standard XML languages. However, a naive implementa-

tion that first generates the canonical document in the left column on p. 16, and then applies the XQuery transformation query, could lead to unacceptable performance, especially for large documents.

### Evaluating Queries over XML Views

XML views hide from the user the details of how the underlying data is stored. For example, the availability of XML views over relational data allows programmers or applications to use XML interfaces and languages to access data in relational tables. At runtime, when XML queries are issued, the publishing system generates and issues one or more SQL queries. The query results (such as a set of relational tables) are then combined and tagged with the appropriate XML markup. Ensuring that these views are efficiently retrieved and queried is an important problem that a publishing system must address. Silkroute,[8] Xperanto,[9] and RoleX[10] are research systems that optimize SQL query generation. Whereas Silkroute and Xperanto output serialized XML in response to XML queries, RoleX provides a virtual DOM interface to relational data.

## Pitfalls, Limitations and Open Problems

XDM is still immature, and many issues have not yet been addressed properly. In this section, we discuss limitations of existing solutions as well as some open problems. Our discussion is biased toward problems we have encountered in trying to create effective and scalable XDM solutions; it is by no means exhaustive.

### The Cost of Parsing and Validation

Parsing and validating a document against an XML Schema or DTD are CPU-intensive tasks that can be a major bottleneck in XML management. A recent study of XML parsing and validation performance indicates that response times and transaction rates over XML data cannot be achieved without significant improvements in XML parsing technology.[11] It suggests enhancements such as using parallel processing techniques and preparsed binary XML formats as well as better support for incremental parsing and validation. Improved performance can also be achieved through application- (schema-) specific parsers, which tools such as XMLBooster (www.xmlbooster.com) can automatically generate.

### Benefits of Compression

Because schema information repeats for every record in a document, XML is an inherently verbose format. To improve the document-exchange performance and reduce storage requirements, it might be advantageous to compress the document. By using XML-specific compression techniques, tools such as XMill[12] compare favorably against several generic compressors. Compression techniques have also been proposed that support direct querying over the compressed data,[13] which besides saving space, also improve query processing times.

### Query Processing and Optimization

There has been an emphasis in XML standards on getting expressive declarative languages, as opposed to languages that we can evaluate efficiently. Languages such as XPath and XQuery provide very succinct ways of expressing hard-to-implement and expensive queries (such as queries with recursion or those that must respect document order). In addition, because complexity can be hidden in a mapping (as in storage or publishing), a simple user query can generate complex queries at the data source. Thus, the topic of XML optimization is arguably more critical and problematic than for relational systems.

*To improve the document-exchange performance and reduce storage requirements, it might be advantageous to compress the document.*

Unfortunately, optimization is still poorly understood. XQuery and XSLT are very broad and combine features from procedural languages with powerful declarative constructs. Hence, almost any optimization technique from either procedural languages or database query languages is a priori applicable. Optimizations that are specific to XML are also possible. XML-specific techniques fall into three broad classes: *schema-based* optimization simplifies queries using DTDs or XML Schemas—this is often done based on rewriting queries;[14] *lower-level* optimizations include the design of XML-specific operators such as specialized join operators,[15] and the use of more *restricted languages* that allow for special-purpose global evaluation algorithms, for example XPath 1.0 evaluation algorithms.[16]

### Querying and Updating Virtual XML Views

As discussed in the "Publishing XML" section, XML views play a key role in XDM. Unlike traditional database views, materializing XML views is not the norm. Materializing a large view to evaluate a selective query, or to perform a single update, can be inefficient. To avoid intermediate materializations, researchers have proposed query composition algorithms;[8] these take an end-user XQuery or XPath request, and the definition of a published view as inputs, and output a new (composed) view definition. A relational query engine can then evaluate the composed view directly over the stored data. The problem of updating XML views over relational databases has recently received attention.[17]

*The ability to support updates is becoming increasingly important as XML evolves into a universal data-representation format.*

### Querying Streams

An important class of XML data comes from streams such as sensor data, stock quotes, and news reports. Queries over these data often perform some sort of filtering—for example, selecting records that match a user-defined condition. Because the data stream is continuous and possibly infinite, queries must be processed in a single pass as the data is parsed. Several techniques have been proposed to efficiently evaluate XPath expressions over streamed XML data, but these techniques have limitations with respect to the class of queries they support—for example, YFilter[18] supports XPath expressions that only contain forward axes (such as child and descendant).

### Updating XML Data

The ability to support updates is becoming increasingly important as XML evolves into a universal data representation format. Although proposals for defining and implementing updates have emerged,[19,20] a standard has yet to be defined for an update language.

An important problem regarding updates is ensuring that a document remains schema-conformant. For some applications, it might be possible to revalidate documents periodically. However, revalidation can be prohibitively expensive if documents are large or if the updates are frequent. Researchers have studied the problem of incremental schema validation of XML data in native format and proposed efficient solutions.[21,22] However, the problem of incremental validation of XML data mapped into relations remains open.

### XML Support in RDBMS

Although XML support in commercial relational engines is improving rapidly, there is a wide variation in the supported features. Some practical problems include proprietary solutions, lack of flexibility, and scalability. Consider, for example, storage mapping. To define a storage strategy, IBM's DB2 XML Extender requires users to write a Document Access Definition specification; consequently, developers must learn a new language to use DB2 (and only DB2) as a backend. The mapping facilities provided by Oracle 9iR2 are not flexible enough to specify many useful mapping strategies. SQLServer's OpenXML requires that documents be compiled into an internal DOM representation, which greatly limits scalability.

### XDM Standards Gap

While standards have been defined for basic XML technology, they are lacking in XDM. No standards exist for defining either publishing or storage mappings, and database vendors have adopted proprietary solutions for both problems that are often limited (for example, not all mapping schemes can be expressed). Efforts are underway in the research community to find a universal mapping framework that encompasses all mapping strategies. ShreX[23] is free system that provides the first comprehensive solution to the relational storage of XML data: it supports a wide range of XML-to-relational mapping strategies, provides generic query translation and document-shredding capabilities, and works with virtually any RDBMS.

### Unknown Performance Characteristics

Although the research community has designed benchmarks such as XBench (http://db.uwaterloo.ca/ddbms/projects/xbench) and XMark (www.xml-benchmark.org), to date, there has been no comprehensive evaluation and performance study of different XDM tools and systems. Hence, it is not clear currently how the various XDM solutions perform, or how scalable they are.In fact, a recent study of XPath evaluation performance[16] uncovered serious inefficiencies in popular XPath processors.

Although existing solutions are evolving, and XML support in commercial products are improving at a fast pace, because XML is so flexible and extensible, we cannot expect to find out-of-the-box XDM solutions for all different applications. Due to the evolving standards, immaturity of the existing tools and the broad scope of the problem, selecting the right system or combination of systems that have the right set of features and meet the performance requirements of a given XML-based application is a nontrivial task. It is thus important that users of this technology be aware of its limitations and avoid known pitfalls.

## Acknowledgments

*The US National Science Foundation partially supports Juliana Freire under grant EIA-0323604.*

## References

1. A. Deutsch, M. Fernandez, and D. Suciu, "Storing Semi-Structured Data with STORED," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD '99), ACM Press, 1999, pp. 431–442.
2. D. Florescu and D. Kossman, "Storing and Querying XML Data Using an RDMBS," *IEEE Data Eng. Bulletin*, vol. 22, no. 3, 1999, pp. 27–34.
3. P. Bohannon et al., "From XML Schema to Relations: A Cost-Based Approach to XML Storage," *Proc. Int'l Conf. Data Eng.* (ICDE '02), IEEE Press, 2002, pp. 64–75.
4. R. Kaushik et al., "Covering Indexes for Branching Path Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD '02), ACM Press, 2002, pp. 133–144.
5. Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proc. 27 th Int'l Conf. Very Large Databases* (VLDB '01), Morgan Kaufmann, 2001, pp. 361–370.
6. H.V. Jagadish et al., "TIMBER: A Native XML Database," *IEEE Data Eng. Bulletin*, vol. 11, no. 4, 2002, pp. 274–291.
7. T. Fiebig et al., "Anatomy of a Native XML Base Management System," *Int'l J. Very Large Data Bases*, vol. 11, no. 4, Springer-Verlag, 2002, pp. 292–314.
8. M.F. Fernandez et al., "A Framework for Publishing Relational Data in XML," *ACM Trans. Database Systems*, vol. 27, no. 4, 2002, pp. 438–493.
9. J. Shanmugasundaram et al., "Querying XML Views of Relational Data," *Proc. 27th Int'l Conf. Very Large Databases* (VLDB '01), Morgan Kaufmann, 2001, pp. 261–270.
10. P. Bohannon et al., "Optimizing View Queries in ROLEX to Support Navigable Result Trees," *Proc. 28th Int'l Conf. Very Large Databases* (VLDB '02), Morgan Kaufmann, 2002, pp. 119–130.
11. M. Nicola and J. John, "XML Parsing: A Threat to Database Performance," *Conf. Information and Knowledge Management*, ACM Press, 2003, pp. 175–178.
12. H. Liefke and D. Suciu, "XMill: An Efficient Compressor for XML Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD '00), ACM Press, 2000, pp. 153–164.
13. P. Tolani and J. Haritsa, "XGRIND: A Query-Friendly XML Compressor," *Proc. IEEE Int'l Conf. Data Eng.* (ICDE '02), IEEE Press, 2002, pp. 225–234.
14. R. Krishnamurthy, R. Kaushik, and J.F. Naughton, "XML-SQL Query Translation Literature: The State of the Art and Open Problems," *Proc. XML Database Symp.* (XSym '03), Springer-Verlag, 2003, pp. 1–18.
15. N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD '02), ACM Press, 2002, pp. 310–321.
16. G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," *Proc. 28th Int'l Conf. Very Large Data Bases* (VLDB '02), Morgan Kaufmann, 2002, pp. 95–106.
17. V.P. Braganholo, S.B. Davidson, and C.A. Heuser, "On the Updatability of XML Views Over Relational Databases," *Proc. Int'l Workshop on Web and Databases* (WebDB), 2003, pp. 31–36; www.cse.ogi.edu/webdb03/.
18. Y. Diao et al., "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," *ACM Trans. Database Systems*, vol. 28, no. 4, 2003, pp. 467–516.
19. I. Tatarinov et al., "Updating XML," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD '01), ACM Press, 2001, pp. 413–424.
20. P. Lehti, *Design and Implementation of a Data Manipulation Processor for an XML Query Language*, masters thesis, Fraunhofer Institut für Integrierte Publikations- und Informationssysteme, 2002; www.ipsi.fhg.de/~lehti/diplomarbeit.pdf.
21. Y. Papakonstantinou and V. Vianu, "Incremental Validation of XML Documents," *Proc. Int'l Conf. Database Theory* (ICDT '03), LNCS, Springer-Verlag, 2003, pp. 47–63.
22. D. Barbosa et al., "Efficient Incremental Validation of XML Documents," *Proc. IEEE Int'l Conf. Data Eng.* (ICDE), IEEE Press, 2004, pp. 671–682.
23. F. Du, S. Amer-Yahia, and J. Freire, "A Generic and Flexible Framework for Mapping XML Documents into Relations," to appear in *Proc. 30th Int'l Conf. Very Large Data Bases* (VLDB), 2004.

**Juliana Freire** is an assistant professor at Oregon Health & Science University's Department of Computer Science and Engineering. Previously, she was a research scientist at Bell Laboratories, Lucent Technologies. Her research focuses on extending traditional database technology and developing novel techniques to address new data management problems introduced by emerging applications, in particular, in the scientific domain and enabled by the advent of the Internet. She obtained her PhD and MS in computer science from the State University of New York at Stony Brook, and her BS in computer science from Universidade Federal do Ceara, Brazil. Contact her at juliana@cse.ogi.edu.

**Michael Benedikt** is a technical staff member in the Network Data and Services Department at Bell Laboratories, Lucent Technologies. His research interests include program verification, query analysis and integrity constraint enforcement in databases, query language support for spatial databases, and foundational issues in XML querying, publishing, and data integration. He received his PhD in mathematics from the University of Wisconsin. Contact him at benedikt@research.bell-labs.com.